Using The Lightweight Workflow Engine

I. Introduction

The Lightweight Workflow Engine (LWWE) is a flexible, portable, multipurpose tool for running workflows. Written entirely in Java, the engine can be used either as a stand-alone server or as an embedded component in another application. Workflows executed under the tool are defined with XML files called Workflow Architectures, which describe the task describing the layout and dependencies within workflows. The engine design allows developers to extend the functionality either through scripting (Jython and Groovy) or with Java.

II. The Workflow Architecture File Structure

Every workflow must at least contain an XML file named workflowArchitecture.xml. This file describes to the engine the tasks and their dependencies. The structure of the file is based on a JAXB object (Java Binding for XML), which allows for easy loading and saving of the workflow.

The workflow architecture file consists of two parts. The first part is the header, which contains various attributes about the workflow, such as who created, who submitted it, etc. The second part is the tasks section, which contains the information such as task names, dependencies, andd other related data.

The Workflow File Header

The structure of the file was designed for simplicity. The header of the file contains information about the workflow, such as who created it and modified it. Below is a sample header for a workflow architecture file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<WorkflowArchitecture xmlns="LightweightWorkflowEngine">
    <Name>Test Workflow</Name>
    <Description>A test workflow</Description>
    <Creator>Jane</Creator>
    <CreationDate>1/1/1111</CreationDate>
    <ModifiedBy>Joe</ModifiedBy>
    <ModificationDate>1/1/2222</ModificationDate>
    <SubmittedBy>Jim</SubmittedBy>
    <Version>1.0.0</Version>
    <OverallStatus>Completed</OverallStatus>
    <MD5Checksum>abdd00f34500cb</MD5Checksum>
…
```

The first line is just the XML specification. The second line declares the object (WorkflowArchitecture) and the namespace for this file (always LightweightWorkflowEngine). These two lines are the same in every workflow file. The rest of header is described in the following list:

- Name: This is the name of the workflow. The name can be any string.
- Description (optional): A brief description of the workflow.
- Creator (optional): The original creator of the workflow.
- CreationDate (optional): The original creation date of the workflow.
- ModifiedBy (optional): Who last modified this workflow.
- ModificationDate (optional): When the last modification took place.
- SubmittedBy (optional): Who submitted this workflow to execute.
- Version (optional): The version of this workflow.
- OverallStatus: The overall status of this workflow. The engine sets this field.
- MD5Checksum (optional): A checksum of the file.

Most for the fields in the header are optional, meaning the workflow will run regardless of what is entered. However, it is a good idea to store meaningful information in these fields as it makes it easier to search and maintain the workflows.

The Workflow File Tasks

The rest of the file is comprised of one or more nested ChildTasks. The following is an example of a ChildTasks XML object.

```
<ChildTasks>
    <Name>System Task A</Name>
    <Description>A system task</Description>
    <Information>Some info</Information>
    <RetriesOnFail>0</RetriesOnFail>
    <TaskStatus>Completed</TaskStatus>
    <TaskType>System</TaskType>
    <IterationLimit>0</IterationLimit>
    <Iteration>0</Iteration>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency>"Task A" eq Completed</TaskDependency>
    <TimeDependency>TIME(2/2/2009:12:15:00)</TaskDependency>
</ChildTasks>
```

The properties of a ChildTasks object are described below:

- Name: Every task must have a UNIQUE name that identifies it. Pretty much any string can be used for a name, including names with spaces.

- Description (optional): A brief description of this task.
- Infromation (optional): Any additional information about the task. Unlike Description, tasks and/or the engine dynamically update this field.
- RetriesOnFail (optional): The number of times to retry an aborted task automatically. This is currently ignored.
- TaskStatus: The status of the task. Initially this can be absent or empty, however the workflow engine automatically populates this field when running.
- TaskType: The type of this tasks.. This will be covered more in depth later.
- IterationLimit (optional): The maximum number of times this task can be executed. A missing value or value of <= 0 indicates that there is no limit. See the Task Looping section for more information.
- Iteration (optional): The current number of times this task has been executed. See the Task Looping section for more information.
- ExecutableObject: The actual object executed by this task when the task and time dependencies are satisfied. Depending on the task type this can be anything from a command line specification to a Java class name.
- TaskDependency (optional): A logical “equation” to evaluate that contains task dependencies for this task. This is covered in depth later.Task Dependency section.
- TimeDependency (optional): A time dependency specification for this task. This is covered laterin deth in the Time Dependency section.

Task Type

The first important field is TaskType. There are several different task types within the engine.

The simplest task type is System, as shown in the example. A System task is a fully automated task that runs whatever command is entered in the ExecutableObject field. In the example, this is simply a command line call to ls, though it could be anything that can be run on the command line, including bash and python scripts, for example.. The engine handles all task status setting and updating. For an initial attempt at a workflow, the System type is the easiest to use.

The other task types (Java, Jython, and Groovy) allow for programmatic customization of a task. These will be covered in the API documentation.

Executable Object

The ExecutableObject for a System task is just a command line operation. It can run anything the normal command line will let you, assuming you have permission to do so. For most workflows, some sort of script execution would be specified here.

Task Dependency

The TaskDependency is an optional field that is the logic equation used to determine when a task may be executed during a workflow. It is an optional field containing a logic equation that determines whether or not all task dependencies have been met for this task. If there is no string specified, then the task will run when the workflow is started (assuming it doesn't have a time dependency).

The grammar for the task dependency string this is simple. Typically, you want to check on task status conditions for various tasks and execute a task when those conditions are met. For example, you may want Task B to run only if Task A has been completed successfully, as shown in the example above. The equation for this is simply:

```
"Task A" eq Completed
```

The quotes around the task name are necessary if your task name contains spaces. **Note that all task names, operators, and status conditions are case sensitive**. For testing task conditions, there are two acceptable operators:

- eq: Checks to see if the status of the specified task is equal to the specified status.
- neq: Checks to see if the status of the specified task is not equal to the status specified.

The engine also supports logical concatenation operators for performing more than one dependency check:

- and: Performs a logical and between two conditions
- or: Performs a logical or between two conditions
- xor: Performs a logical exclusive or between two conditions

The logical concatenators are for evaluating multiple dependencies. For example, a task with two dependencies may look like:

```
"Task B" eq Completed and "Task A" eq Completed
```

You can also use parentheses to set the order of operations:

```
"Task C" eq Completed or ("Task A" eq Completed and "Task B" eq Completed)
```

To the engine, this reads as “If Task C is completed or if Task A and Task B are completed.

The task status conditions can be any of the following:

- Completed: The task completed successfully
- Failed: The task failed
- Aborted: The task was aborted
- Suspended: The task was suspended
- Resumed: The task was resumed
- Updated: The task was updated
- Queued: The task was queued
- Running: The task was running

So another task dependency may look like this:

```
"Task B" eq Completed or "Task A" eq Failed
```

Which can be read as "Run this task if Task B is completed or if Task A failed".

Time Dependency

In addition to a task dependency, there can also be a time dependency. Time dependencies are instigated after task dependencies have been satisfied. The format here is also simple. Here is an example:

```
TIME(2/2/2009:12:15:00)
```

All time dependencies have the format TIME(...). A time dependency can have up to three arguments. The following are the acceptable time dependency specifications:

- TIME (datetime): Specifies a time dependency that will fire at the specified date and time.
- TIME (datetime, interval): Specifies a time dependency that begins at a specified date and re-triggers after every interval.
- TIME (datetime, interval, delay): Specifies a time dependency that begins at a specified date. After the delay, it will re-trigger at the specified interval.

Using * for the datetime will automatically use the current system time. So a time dependency specified like this:

```
TIME(*, 3600)
```

Would set up a time dependency that would repetitively trigger after every hour, using the current time as the starting point.

Both Task and Time dependencies are optional. If no dependencies are specified, then the task will trigger whenever the workflow is started.

Nested Tasks

There are no limits to the depth of nesting for tasks. However, in practice having heavily nested tasks can lead to an unwieldy file.

While XML lends itself to a hierarchical structure, that structure is superficial to the workflow itself. Task execution is determined by dependencies, not their placement within the workflow file or how deeply they are nested. However, it is a good practice to place tasks where they logically would occur during workflow execution.

A nested task would look similar to this:

```
<ChildTasks>
    <Name>System Task A</Name>
    <Description>A system task</Description>
    <TaskStatus>Completed</TaskStatus>
    <TaskType>System</TaskType>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency>”Task A” eq Completed</TaskDependency>
    <TimeDependency>TIME(2/2/2009:12:15:00)</TaskDependency>
    <ChildTasks>
     <Name>System Task B</Name>
     <Description>A system task</Description>
     <TaskStatus>Completed</TaskStatus>
     <TaskType>System</TaskType>
     <ExecutableObject>ls -lrt</ExecutableObject>
     <TaskDependency>”Task A” eq Completed</TaskDependency>
     <TimeDependency>TIME(2/2/2009:12:15:00)</TaskDependency>
    </ChildTasks>
</ChildTasks>
```

Task Looping

Task looping is handled by specifying task dependencies in such a way as to cause a loop. For example:

```
<ChildTasks>
    <Name>Task A</Name>
    <Description>A task</Description>
    <TaskStatus></TaskStatus>
    <TaskType>System</TaskType>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency></TaskDependency>
</ChildTasks>
<ChildTasks>
    <Name>Task B</Name>
    <Description>A looping task</Description>
    <TaskStatus></TaskStatus>
    <TaskType>System</TaskType>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency>"Task A" eq Completed or "Task C" eq
Completed</TaskDependency>
    <ChildTasks>
      <Name>Task C</Name>
      <Description>Another task</Description>
      <TaskStatus></TaskStatus>
      <TaskType>System</TaskType>
      <ExecutableObject>ls -lrt</ExecutableObject>
      <TaskDependency>"Task B" eq Completed</TaskDependency>
    </ChildTasks>
</ChildTasks>
```

As can be seen, Task A has no dependencies, so starts when the workflow is executed. Task B will trigger whenever Task A is completed or whenever Task C is completed. Task C is triggered whenever Task B is completed.

When the workflow runs, Task A will execute to completion. Barring any errors, it will notify Task B that it has completed. This will trigger Task B to run. When Task B completes, it will notify Task C. When Task C completes, it will again notify Task B to run.

An even simpler loop can be made:

```
<ChildTasks>
    <Name>Task A</Name>
    <Description>A task</Description>
    <TaskStatus></TaskStatus>
    <TaskType>System</TaskType>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency>"Task A" eq Completed</TaskDependency>
</ChildTasks>
```

Task A depends only on itself completing, and will continuously retrigger every time it completes.

However, this type of looping is infinite. There are no limits on how many times the task will be called. For better control, a workflow designer would use the IterationLimit field.

```
<ChildTasks>
    <Name>Task A</Name>
    <Description>A task</Description>
    <TaskStatus></TaskStatus>
    <TaskType>System</TaskType>
    <IterationLimit>10</IterationLimit>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency>”Task A” eq Completed</TaskDependency>
</ChildTasks>
```

By specifying the iteration limit, the engine will only execute a task up to the iteration limit. After that, the task remains in its completed state and no longer sends out triggering information unless forced to by a user action.

The current iteration of a task is tracked ONLY if the IterationLimit has been set. The field used for tracking the current iteration is:

```
<Iteration>3</Iteration>
```

So within a workflow file, and iterative task may look like this:

```
<ChildTasks>
    <Name>Task A</Name>
    <Description>A task</Description>
    <TaskStatus></TaskStatus>
    <TaskType>System</TaskType>
    <IterationLimit>10</IterationLimit>
   <Iteration>3</Iteration>
    <ExecutableObject>ls -lrt</ExecutableObject>
    <TaskDependency>”Task A” eq Completed</TaskDependency>
</ChildTasks>
```

This field is used by the engine, and is stored with the rest of the workflow. While manipulation of this field is possible, it is not recommended.

System Task Logging

All system type tasks automatically create a log file which stores all stderr and stdout output from the executing process.

Other task types are free to implement their own form of logging, or none at all if it is deemed unnecessary.

Updating System Task Information With Messages

As noted, a System type task is a special task that runs a command like a command line. The call to the system is independent of the workflow engine. Other than starting the process and analyzing the return code, there is no direct interaction with the running task.

However, occasionally there may be need to have more information than just the state of a task to be sent to a user or stored with the workflow. In these instances, a workflow designer can have a task output messages with special keywords to indicate to the engine that the user should be notified or the information should be stored.

The keywords **MUST** appear at the start of the line and be terminated with a colon character. The keywords are case sensitive. The following is a list of keywords, their meanings, and how the engine handles the message.

- FATAL: Indicates a fatal problem. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- ERROR: Indicates an error. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- WARN: Indicates a warning. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- INFO: Indicates an information message. The string is used to create a status message, which is sent back to anyone monitoring the workflow.
- DEBUG: A debugging message. The string is used to create a status message, which is sent back to anyone monitoring the workflow. These should be removed before a release.
- UPDATE: Indicates that a task's information should be updated. Unlike the other keywords, this does not create a status message. Instead, the information is used to update the task information. This information is stored with the workflow.

Some examples of parse-able messages are:

```
UPDATE: I'm running my second loop!
ERROR: No directory was found!
WARN: Value X was 3 but should have been 4.
```

III. Workflow Execution

When a workflow architecture file is loaded by the engine, it first goes through and loads all the tasks. Then the engine goes through each task, determines the dependencies, sets up the necessary notifications, and waits to receive a run command.

When a run command is received, the engine broadcasts a start message to the workflow. Any tasks that have no dependencies are immediately triggered to run. **In every workflow there must be at least one task that has no dependencies, otherwise the workflow will not be able to execute**.

An engine user may also set "environment" values into the engine. These values are accessible directly via the API for Java, Groovy, and Jython tasks. For System tasks, these variables are added as process environment variables. This functionality is useful for setting common variable values that are used throughout the workflow, such as a user name or a group code. The System type tasks however are not able to alter these variables other than in its own process.

IV. Appendix

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<WorkflowArchitecture xmlns="LightweightWorkflowEngine">
    <Name>LWWE_null</Name>
    <Description>A workflow for testing the workflow engine</Description>
    <Creator>ME</Creator>
    <CreationDate>1/1/1111</CreationDate>
    <ModifiedBy>YOU</ModifiedBy>
    <ModificationDate>1/1/2222</ModificationDate>
    <SubmittedBy>someone</SubmittedBy>
    <Version>1.0.0</Version>
    <OverallStatus></OverallStatus>
    <MD5Checksum>0</MD5Checksum>
    <ChildTasks>
        <Name>Task A</Name>
        <Description>A system task</Description>
        <TaskStatus></TaskStatus>
        <TaskType>System</TaskType>
        <ExecutableObject>ls -lrt</ExecutableObject>
        <TaskDependency></TaskDependency>
    </ChildTasks>
    <ChildTasks>
        <Name>Task B</Name>
        <Description>A system task</Description>
        <TaskStatus></TaskStatus>
        <TaskType>System</TaskType>
        <ExecutableObject>echo "Task B says hello"</ExecutableObject>
        <TaskDependency>"Task A" eq Completed</TaskDependency>
       <!–This is a child task of task b →
       <ChildTasks>
            <Name>Task C</Name>
            <Description>A system task</Description>
            <TaskStatus></TaskStatus>
            <TaskType>System</TaskType>
            <IterationLimit>10</IterationLimit>
            <ExecutableObject>echo "Task C says hello repeatedly!"</ExecutableObject>
            <TaskDependency>"Task B" eq Completed or "Task C" eq
```

```
Completed</TaskDependency>
        </ChildTasks>
    </ChildTasks>
</WorkflowArchitecture>
```